BinarySearchTree

Что такое бинарное дерево поиска подробно описано в 8-м занятии V семестра, не буде здесь повторять этот текст, там всё просто, раве что для формальной полноты картины приведу цитату из Вики - `https://ru.wikipedia.org/wiki/Двоичное_дерево_поиска` .
цитата:
Для целей реализации двоичное дерево поиска можно определить так:

> - Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.
> - Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
> - Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и не строго меньше ключей данных правого.

конец цитаты

Бинарное дерево поиска. Простые операции: добавление/ поиск элемента, обход дерева.

Для начала, конечно объявим типы: тип узла дерева поиска и собственно тип дерева.

```
type (
    Node struct {
        Key  order.Key
            // интерфейс, включающий в себя сравнение
            // с другим ключом и визуализацию ключа
        Lson Tree
        Rson Tree
    }
```

```
    Tree struct {
        Root *Node
    }
)
```

Вот они описания интерфейсов.

```
type Ordered interface {
    Before(b Ordered) bool
}

type Visual interface {
    Show() string
}

type Key interface {
    Ordered
    Visual
}
```

Давайте для начала посмотрим на три простых метода: Insert, который добавляет новый ключ в дерево поиска, Find, проверяющий наличие заданного ключа в дереве поиска, и Traversal, обходящий дерево и возвращающий слайс со значениями ключей дерева в порядке возрастания или убывания. Но просто топтаться на том же месте скучно, давайте немного пообобщаем, совсем немного.

Во-первых, добавим различные порядки обхода - кроме InOrder'а (прямой InOrder для вывода элементов в порядке возрастания ключей и обратный - ReverseInOrder - для вывода в порядке убывания), включим Preorder (прямой и обратный) и PostOrder (тоже прямой и обратный).
Пару слов о применениях этих обходов. *PreOrder* замечательно подходит для хранения дерева, для создания копии. В самом деле, если мы, например, запишем ключи в *PreOrder*'е, то при вставке ключей в новое дерево, мы получим в отчности такое же дерево, как и исходное. Соответственно, *PostOrder* хорошо ложится на процедуру удаления дерева - всё получается корректно. Понятно, что *Go*-шный сборщик мусора прекрасно сам справляется, но всякие ситауции бывают.

Во-вторых, вынесем всю работу непосредственно с ключами в методы типа ключа - соответственно появился интерфейс `Key` .

Всё это отражено в следующем коде:

```
func (t Tree) Empty() bool {
    return t.Root == nil
}

func (t *Tree) Insert(value order.Key) {
    if (*t).Empty() {
        (*t).Root = &Node{Value: value, Lson: NewTree(), Rson: NewTree()}
        return
    }
    if value.Before((*t.Root).Value) {
        // value "<"" (*t.Root).Value
        (*t.Root).Lson.Insert(value)
    } else {
        // Value "!<" (*t.Root).Value {
        (*t.Root).Rson.Insert(value)
    }
}

func (t Tree) Find(n order.Ordered) bool {
    switch {
    case t.Empty():
        return false
    case n.Before((*t.Root).Value):
        //    n before (*t.Root).Value
        return (*t.Root).Lson.Find(n)
    case (*t.Root).Value.Before(n):
        //    n after (*t.Root).Value
        return (*t.Root).Rson.Find(n)
    default:
        // n is equivalent to (*t.Root).Value
        return true
    }
}

func (t Tree) Traversal(dir int, f func(x order.Key)) {
    if t.Empty() {
```

```
        return
    }
    switch dir {
    case order.PreOrder: // NLR
        f((*t.Root).Value)
        (*t.Root).Lson.Traversal(dir, f)
        (*t.Root).Rson.Traversal(dir, f)
    case order.InOrder: // LNR
        (*t.Root).Lson.Traversal(dir, f)
        f((*t.Root).Value)
        (*t.Root).Rson.Traversal(dir, f)
    case order.PostOrder: // LRN
        (*t.Root).Lson.Traversal(dir, f)
        (*t.Root).Rson.Traversal(dir, f)
        f((*t.Root).Value)
    case order.ReversePreOrder: // NRL
        f((*t.Root).Value)
        (*t.Root).Rson.Traversal(dir, f)
        (*t.Root).Lson.Traversal(dir, f)
    case order.ReverseInOrder: // RNL
        (*t.Root).Rson.Traversal(dir, f)
        f((*t.Root).Value)
        (*t.Root).Lson.Traversal(dir, f)
    case order.ReversePostOrder: // RLN
        (*t.Root).Rson.Traversal(dir, f)
        (*t.Root).Lson.Traversal(dir, f)
        f((*t.Root).Value)
    }
}
```

Бинарное дерево поиска. Удаление элемента.

Задача формулируется просто: удалить из дерева узел с заданным ключом x. При этом необходимо так преобразовать оставшиеся данные, чтобы они по-прежнему образовывали бинарное дерево поиска. Если в дереве поиска имеется несколько элементов с ключами, равными x, то удалить любой из них.

И ещё одно очень важное требование: преобразовать дерево поиска надо как можно

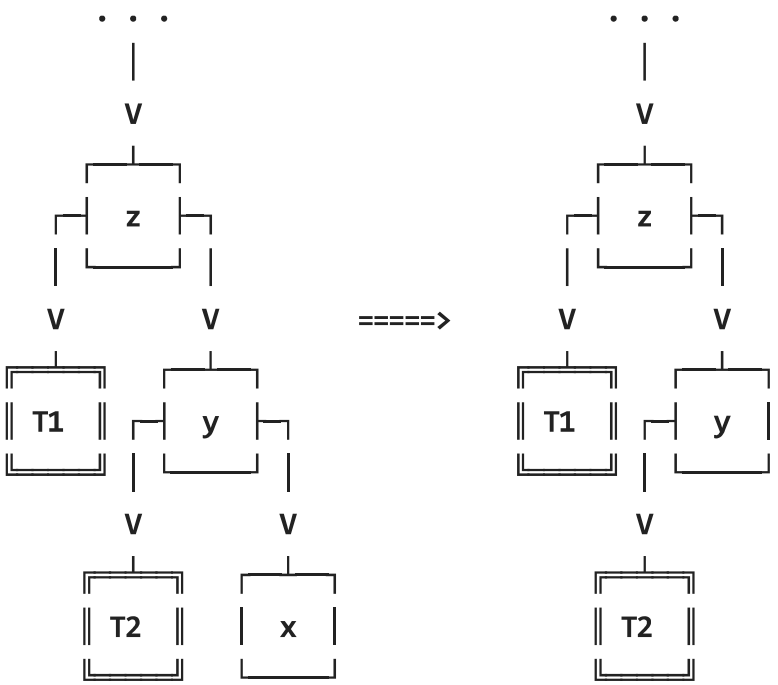
более эффективно, например, вариант собрать все остающиеся элементы и построить из них новое дерево поиска не прокатывает абсолютно.

Начнём с самого простого случая: удалить лист дерева (вершину, у которой нет потомков).

Решение очевидное - просто удаляем этот узел, делаем соответствующий указатель его родителя равным nil. Очевидно, что оставшаяся структура является деревом поиска. На рисунке всё видно.

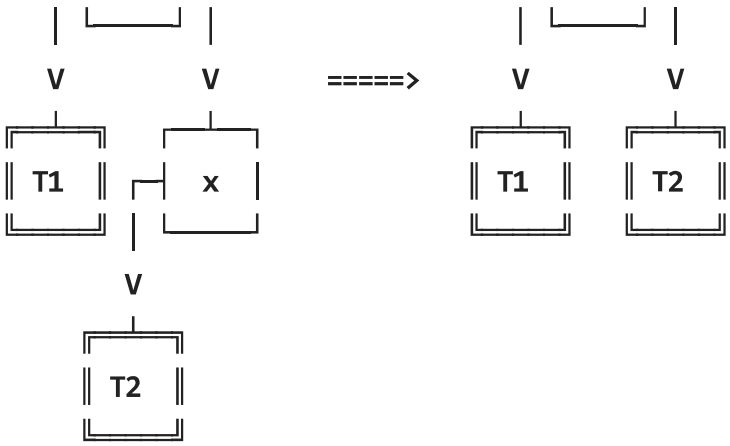
Примечание. Мы будем на схемах изображать узлы прямоугольниками с одинарной границей, а поддеревья - прямоугольниками из двойных линий.

На рисунке мы видим три узла, содержащих ключи x, y и z и два подерева T1 и T2.



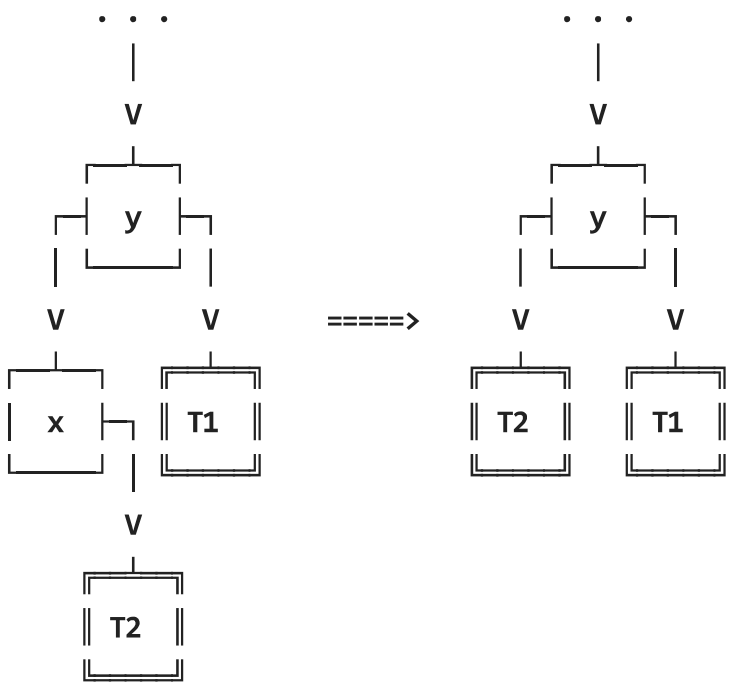
Следующий случай: удаляемый узел имеет только одного сына.

И здесь тоже всё легко. Опять просто убираем узел x из дерева. Родитель узла x будет теперь вместо ссылки на x содержать ссылку на единственное поддерево узла x.



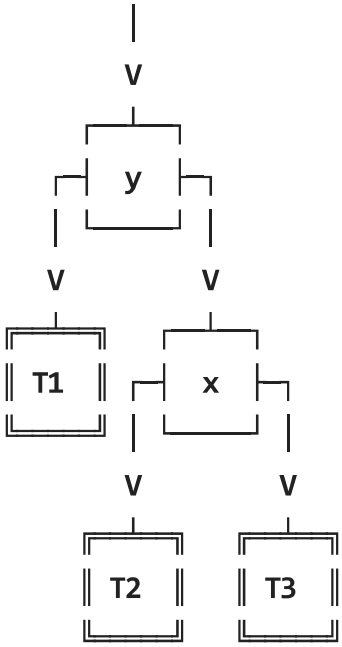
Заметим, что оставшаяся структура является бинарным деревом поиска. В самом деле, все элементы поддерева T2 попали в правое поддерево узла y, т.е. все они >= y - условие дерева поиска сохраняется. И совершенно неважно, было привешено поддерево T2 к узлу x слева или справа.

Если бы узел x был бы левым сыном, то всё аналогично, только все элементы поддерева T2 были бы меньше y. Вот рисунок:



Заметим, что этот случай включает в себя и первый случай - удаление листа. Действительно, можем считать что лист имеет только одного потомка, да и тот пустой.

А вот теперь **самый интересный случай**: удаляемый узел имеет двух сыновей:



И тут сразу видно, что просто так удалить узел x не получится: тогда получится, что у узла y будет три сына, так нельзя. Что же делать? И решение есть: заменить узел x на кого-то из его потомков. На какой? Не будем ходить вокруг да около, сразу скажем: либо на наибольший элемент левого поддерева T2, либо на наименьший элемент правого поддерева T3. Давайте остановимся на втором варианте, тому есть некоторые обоснования, но о них потом. Итак, план такой: найти наименьший элемент в поддереве T3, поставить его вместо x, а из дерева T3 его удалить. Немножко настораживает необходимость снова что-то там удалять, но сейчас-то мы уже знаем, что сделать это можно рекурсивно, и это легко. Сразу замечу, что сделать это оказывается ещё легче, чем кажется. но сначала убедимся, что указанное преобразование сохранит свойства дерева поиска. Ну, то, что оно останется бинарным деревом очевидно. Далее. Все элементы поддерева T3 не меньше x, тем более все они больше любого элемента поддерева T2, так что с левым поддеревом всё в порядке. В то же время на место x мы ставим наименьший элемент поддерева T3, так что и с правым поддеревом всё в порядке.

Остаётся найти наименьший элемент в поддереве T3. Но это совсем легко - просто двигаемся от корня поддерева влево до упора. Замечательно. У наименьшего элемента поддерева нет левого сына, удалить его из поддерева T3 совсем легко - он же попадает под предыдущий случай. Вот и всё.

Резюмируя: рекурсивно доходим до узла с ключом x, если такой есть (если его нет, то делать ничего вообще не надо) и заменяем его на наименьший элемент правого поддерева этого узла. Описанные действия дословно переведены в код:

```
func (t *Tree) Delete(value order.Ordered) {
    if t.Empty() {
        return
    }
    switch {
    case value.Before((*t.Root).Value):
        // value "<" (*t.Root).Value
        (*t.Root).Lson.Delete(value)
    case (*t.Root).Value.Before(value):
        // value ">" (*t.Root).Value:
        (*t.Root).Rson.Delete(value)
    default:
        // value "==" (*t.Root).Value:
        switch {
        case (*t.Root).Lson.Empty():
            *t = (*t.Root).Rson
        case (*t.Root).Rson.Empty():
            *t = (*t.Root).Lson
        default:
            // !(*t.Root).Lson.Empty() && (*t.Root).Rson.Empty()
            min := (*t.Root).Rson.leftmost()
            (*t.Root).Rson.Delete((*min.Root).Value)
            (*min.Root).Lson, (*min.Root).Rson = (*t.Root).Lson, (*t.Root).Rson
            *t = min
        }
    }
}

func (t Tree) leftmost() Tree {
    if t.Empty() {
        return NewTree()
    }
    for !(*t.Root).Lson.Empty() {
        t = (*t.Root).Lson
    }
    return t
}
```

Один фрагмент нуждается в комментариях - это фрагмент, где мы заменяем узел с ключом value на наименьший элемент его правого поддерева. Рассмотрим его поподробнее:

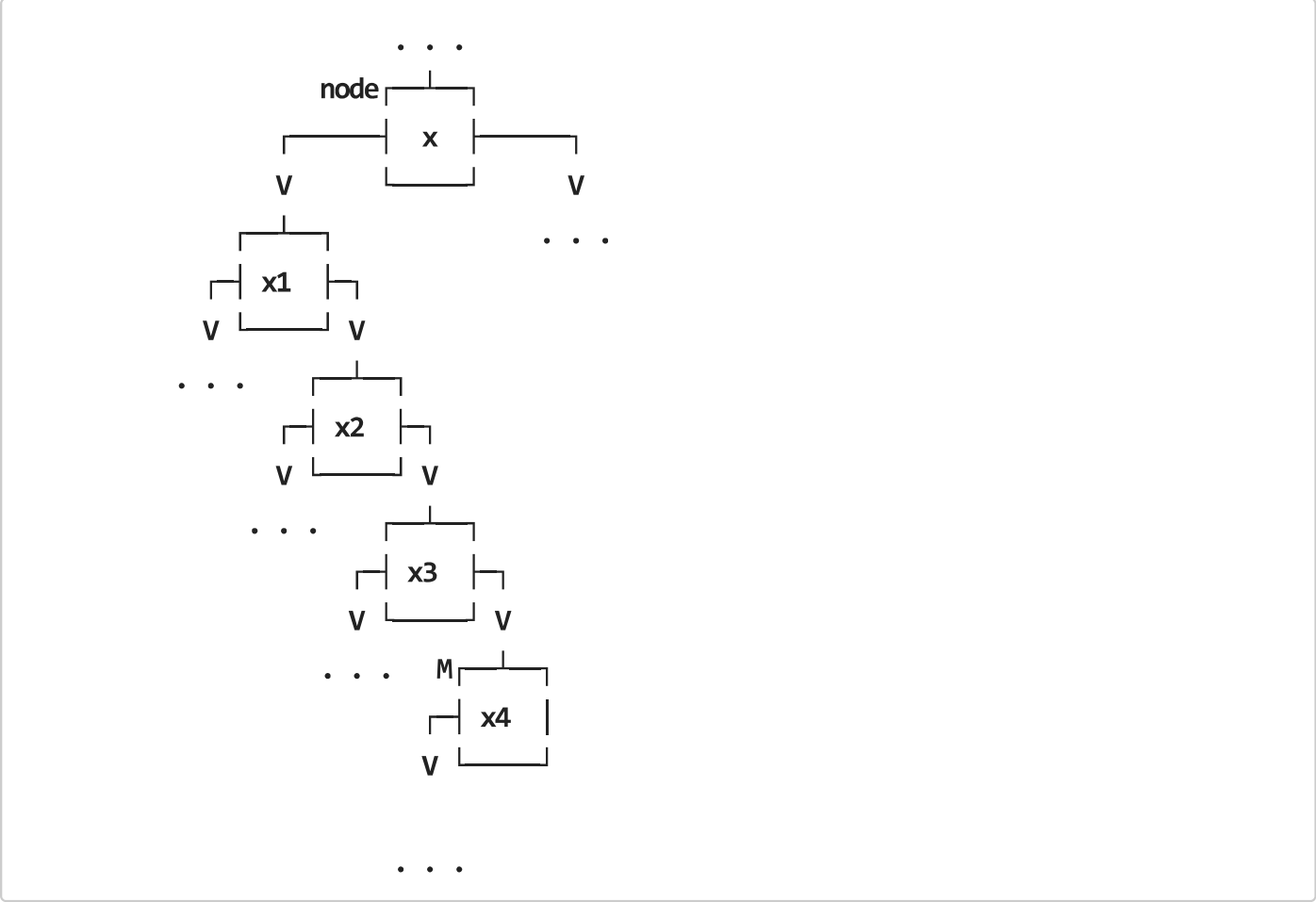
```
1.            // !(*t.Root).Lson.Empty() && (*t.Root).Rson.Empty()
2.            min := (*t.Root).Rson.leftmost()
3.            (*t.Root).Rson.Delete((*min.Root).Key)
4.            (*min.Root).Lson, (*min.Root).Rson = (*t.Root).Lson, (*t.Root).Rson
5.            *t = min
```

```
1.    /* node.lson != nil && node.rson != nil */ {
2.        minNode := leftmost(node.rson)
3.        node.rson = t.delete(minNode.key, node.rson)
4.        minNode.lson, minNode.rson = node.lson, node.rson
5.        node = minNode
6.    }
```

Ключ в корне дерева t node равен value. В строке 2. мы находим узел, содержащий наименьший ключ в правом поддереве дерева t, в строке 3. - удаляем этот узел. И вот здесь есть тонкий момент. Интернет с подавляющим перевесом рекомендует просто заменять значение ключа в корне дерева t на значение наименьшего ключа в его правом поддереве. Это довольно некорректное действие. Представьте себе, что на корень дерева t “смотрит” какой-то внешний (по отношению к дереву поиска) указатель. Ситуация совершенно естественная, мы же собираем значения ключей в дерево поиска для того, чтобы как-то их упорядочить, выяснить какие-то свойства массива ключей (массив в данном случае имеет общечеловеческое значение, а не массив или слайс как структура данных). Т.е. дерево поиска есть служебный инструмент для чего-то, оно редко самоценно само по себе. И теперь представьте себе, как удивится внешний указатель, когда значение его ключа вдруг изменится ни с того ни с сего. Так что мы поступим корректно - заменим корень дерева t на найденный узел min (узел с наименьшим ключом в правом поддереве дерева t), а сам корень (теперь уже бывший корень) просто отпустим, освободим дерево от него, а его оставим в покое. Именно это и исполняют строки 4. и 5. приведённого фрагмента.

И последнее замечание. Почему мы всё-таки выбрали заменять удаляемый узел на узел с наименьшим ключом в правом поддереве, а не на узел с наибольшим ключом в левом поддереве? Дело в том, что ключ в левом сыне строго меньше ключа родителя, а ключ в правом сыне больше или равен родительского ключа. Представьте себе, что

мы решили заменить узел с ключом value на узел с наибольшим ключом в левом поддереве. Этот узел, назовём его M, мы успешно находим, двигаясь по левому поддереву узла node до упора вправо.



И теперь удаляем узел M. Но оказывается, что x3 == x4, и тогда либо мы удалим узел с ключом x3, что некорректно и совершенно неприемлемо - структура испортится безнадёжно, либо при удалении узла придётся применять какие-то дополнительные проверки.

Дополнительные проверки - это не очень страшно, хоть и, пусть совсем чуть-чуть, но замедляет процесс, а мы же боремся за побыстрее, а избранный нами вариант обходится без таких проверок - значение ключа в левом сыне отличается от значения ключа в родителе, оно строго меньше. Это соображение и обосновывает наш выбор.

Вот теперь всё про удаление элемента из дерева поиска.

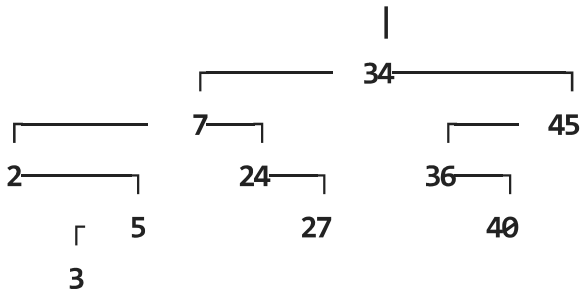
Бинарное дерево поиска. Визуализация.

Сразу покажу, что выводят две наших функции, визуализирующие дерево поиска.

Вывод результата функции DraftDisplay:

```
    45
        40
      36
34
        27
      24
    7
        5
          3
      2
```

Вывод результата функции Diagram:



Функция DraftDisplay нужна здесь только для того, чтобы объяснить основную идею визуализации. Т.е. это такая впомогательная, промежуточная версия функции визуализации. Выбранный ракурс позволяет реализовать фунцию визуализации максимально просто. Идея довольно понятная - на самой левой позиции печатаем ключ в корне дерева поиска. С учётом нашего поворота правое поддерево надо вывести до того, как выведем корень, а левое - после того. Так и сделаем. рекурсивно, разумеется. При этом оба поддерева должны быть сдвинуты вправо, т.е. порядок обхода дерева такой:

- выводим правое поддерево, сдвинутое вправо относительно корня
- выводим корень дерева
- выводим левое поддерево, сдвинутое вправо относительно корня

В примере исполняется сдвиг на 3 позиции. Да, мы не выводим результаты сразу на экран, а собираем их в одну строку. Понятно, что после вывода ключа, надо переходить на следующую строку экрана. Это и достигается присоединением символа перевода строки “\n” сразу вслед за числом. Процедура прозрачная, кажется, больше

комментировать её ни к чему. Просто смотрим код:

```
func (t Tree) DraftDisplay(indent string) string {
    if t.Empty() {
        return ""
    }
    return (*t.Root).Rson.DraftDisplay(indent+"    ") +
        indent + (*t.Root).Value.Show() + "\n" +
        (*t.Root).Lson.DraftDisplay(indent+"    ")
}
```

Восстановить дерево со всеми его связями глядя на результат работы функции DraftDisplay легко: проводим мысленно из самой левой вершины (а это корень) горизонтальную линию, все, что окажется выше неё - правое поддерево, и мы можем просто провести дугу от корня дерева к крайней левой вершине правого поддерева, а затем повторяем процедуру для правого поддерева. Аналогично поступаем с левым поддеревом, которое оказывается ниже мысленно проведённой горизонтальной линии.
Но не будем заниматься развитием этой функции, не будем рисовать соединительные линии, а перейдём к рисованию дерева в привычном развороте.

Понятно, что в привычном ракурсе (так, как это изображено в выводе результата функции Diagram) корень дерева располагается в верхней строке. Проблема здесь в том, что мы не можем определить его местоположение до тех пор, пока не определим ширину изображения левого поддерева. Да, а ещё надо провести соединительную линию от корня дерева к его левому сыну. Потом и к правому, но это потом. В общем идея такая: храним изображение дерева в слайсе строк, причём все строки имеют одинаковую длину, т.е. изображение дерева представляет из себя ровненький такой прямоугольник символов, что-то типа такого (x - это произвольный символ, в том числе, возможно и пробел):

```
        xxxxxxxxxxx
        xxxxxxxxxxx
        xxxxxxxxxxx
        xxxxxxxxxxx
        xxxxxxxxxxx
```

А для того, чтобы подвешивать поддерево к его родителю мы над верхней строкой

прицепим строку такой же длины, в которойвсе символы пробелы, кроме одного - вертикальной чёрточки над корнем дерева (символы ~ здесь означают пробел):

```
~~~~~|~~~~~
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
```

Очень симпатичная картинка получается.

Пустое дерево представляется слайсом из одной пустой строки, строки с вертикальной чёрточки в этом педставлении там нет.

И тогда склейка корня дерева с двумя его поддеревьями получается легко и естественно. Вот есть два поддерева - левое и правое:

```
~~~~~|~~~~~      ~~~|~~~
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
```

Уравниваем их высоты, добавляя к более низкому дереву необходимое количество строк, заполенных необходимым количеством пробелов:

```
~~~~~|~~~~~      ~~~|~~~
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
xxxxxxxxxxx      xxxxxxx
```

Опять-таки, пустое дерево представлялось слайсом из одной пустой строки, соответственно этот слайс дополняется необходимым количеством пустых строк.

И тогда склейка производится в три шага

```
        ~~~~~~~~~~~~~~|~~~~~~~~
        ~~~~~┌─────root─────┐~~~
        xxxxxxxxxxx~~~~xxxxxxx
        xxxxxxxxxxx~~~~xxxxxxx
        xxxxxxxxxxx~~~~xxxxxxx
        xxxxxxxxxxx~~~~xxxxxxx
        xxxxxxxxxxx~~~~xxxxxxx
```

1. верхнюю строку в левом поддереве (если оно непустое) заменяем на строку такой же длины, в которой правая часть `|~~~~` заменяется на `┌────`; верхнюю строку в правом поддереве (если оно непустое) заменяем на строку такой же длины, в которой левая часть `~~~~|` заменяется на `────┐`.
2. все строки слайсов, представляющих левое и правое поддеревья склеиваем, причём между первыми строками слайсов вставляется изображение корня дерева, а между всеми остальными строками вставляется такое же по размеру количество пробелов (на картинке всё выглядит понятнее, чем здесь написано).
3. в начало полученного слайса добавляем новую верхнюю строку из пробелов и одной вертикальной черты над корнем.

И всё получается замечательно. Подробности в коде:

```
func (t Tree) Diagram() []string {
    if t.Empty() {
        return []string{""}
    }
    sL := (*t.Root).Lson.Diagram()
    sR := (*t.Root).Rson.Diagram()
    spaceL := strings.Repeat(" ", stringLen(sL[0]))
    for len(sL) < len(sR) {
        sL = append(sL, spaceL)
    }
    spaceR := strings.Repeat(" ", stringLen(sR[0]))
    for len(sR) < len(sL) {
        sR = append(sR, spaceR)
    }
    r := []rune(sL[0])
    ch := ' '
    for i := range r {
```

```
        if r[i] == '|' {
            r[i] = '┌'
            ch = '─'
        } else {
            r[i] = ch
        }
    }
    sL[0] = string(r)
    r = []rune(sR[0])
    ch = '─'
    for i := range r {
        if r[i] == '|' {
            r[i] = '┐'
            ch = ' '
        } else {
            r[i] = ch
        }
    }
    sR[0] = string(r)

    s := []string{spaceL + fmt.Sprintf("%*s", order.ImageWidth, "|") + spaceR}
    s = append(s, sL[0]+(*t.Root).Value.Show()+sR[0])
    for i := 1; i < len(sL); i++ {
        s = append(s, sL[i]+strings.Repeat(" ", order.ImageWidth)+sR[i])
    }
    return s
}

func stringLen(s string) int {
    return len([]rune(s))
}
```